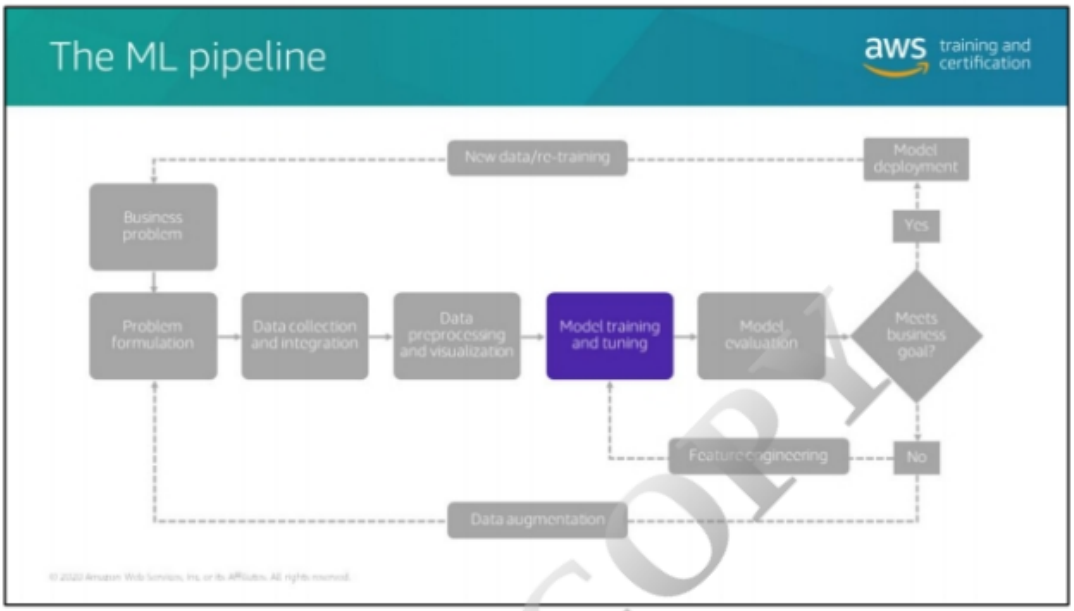
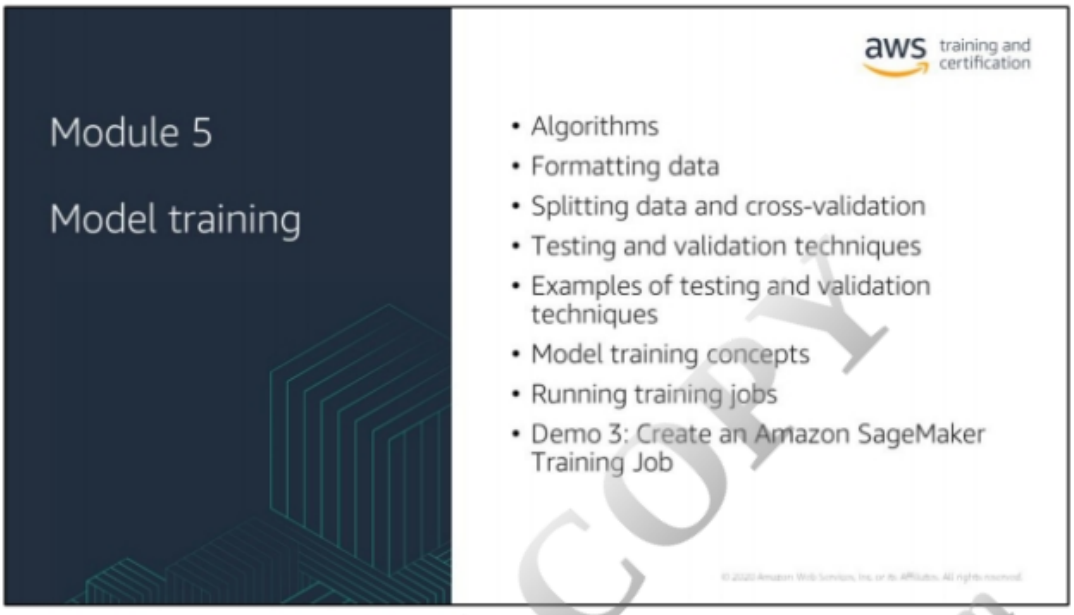
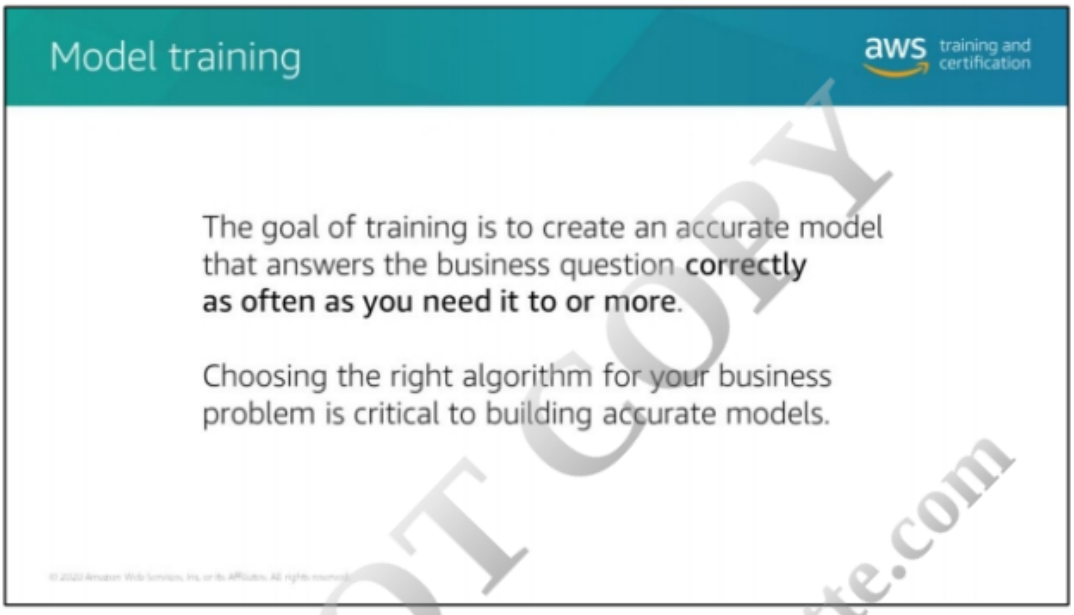
Let's move on to the next phase, model training.



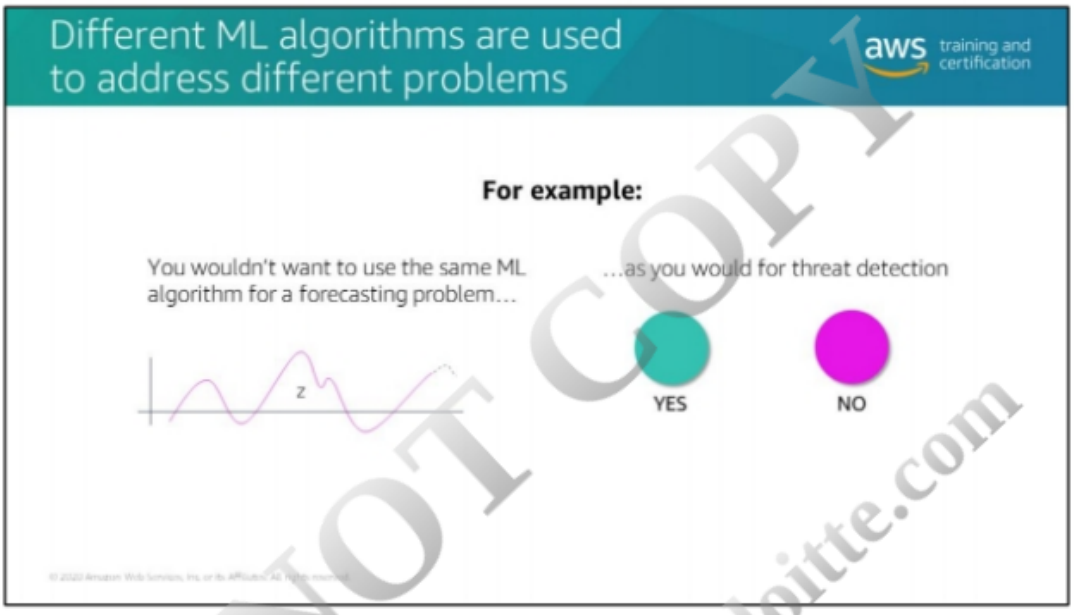
Now that we've explored how to turn business problems into ML problems and have a better understanding of how to define success metrics, examine and choose your data, choose your algorithm, and get your data ready – it's time to train and tune your model.



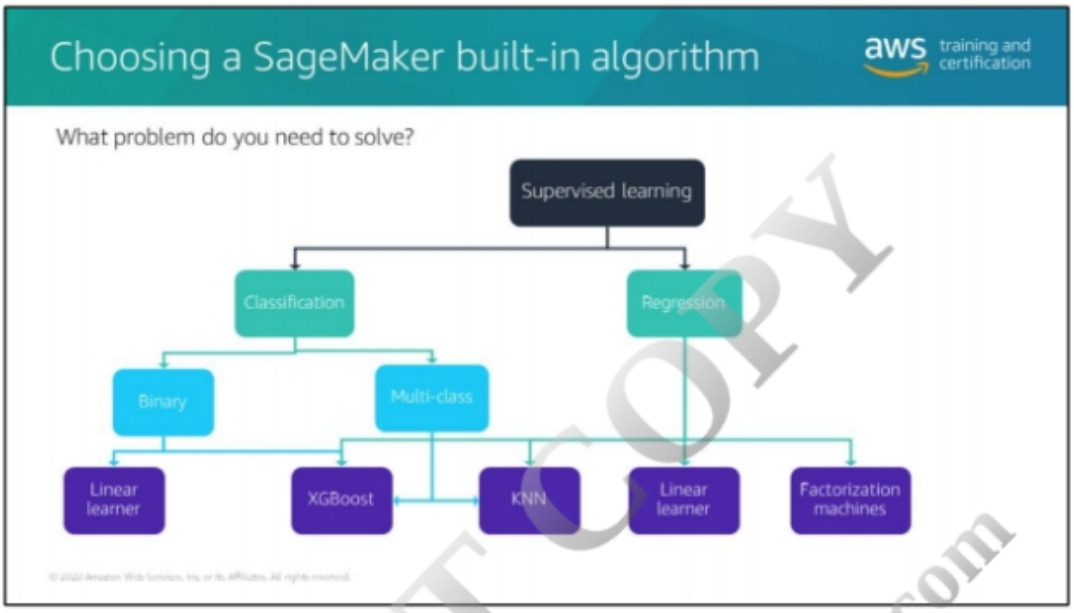
This next module walks you through the model training process. It covers some quick tips for selecting the appropriate algorithm, formatting your data for the algorithm, splitting your data and applying cross-validation methods, and model training and running training jobs. We'll end this module with a demo of how to create an Amazon SageMaker Training Job.



Remember, in the previous module, we transformed raw data into an understandable format. Now we need to use that data to create an accurate model that is capable of answering the business problem correctly MOST of the time. To get this kind of model, we need to select the appropriate algorithm.



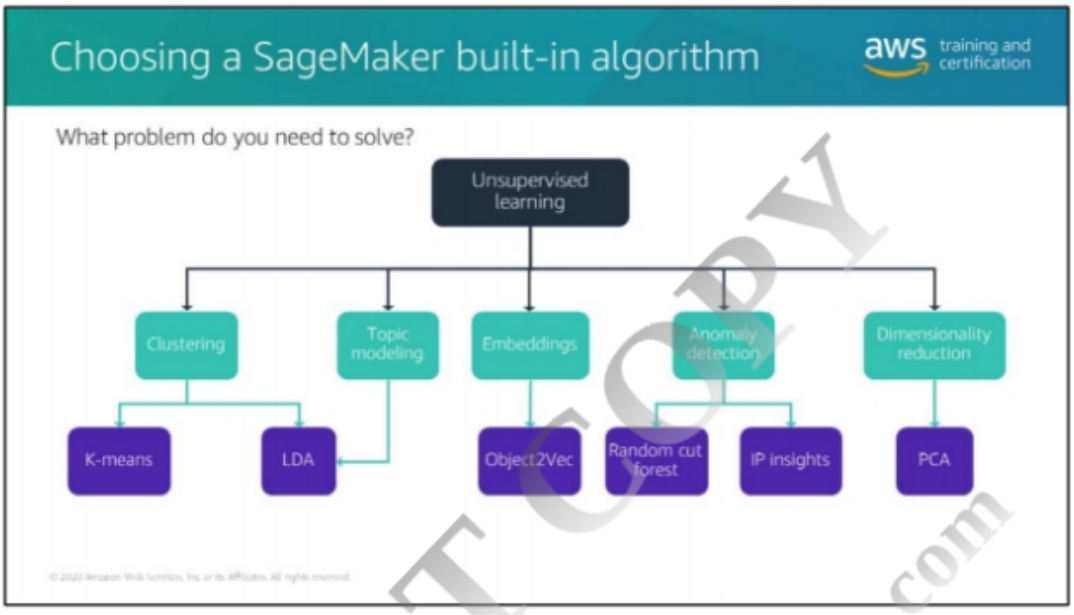
The ML algorithm you choose will heavily depend on what kind of problem you have, as each are designed with certain problems in mind.



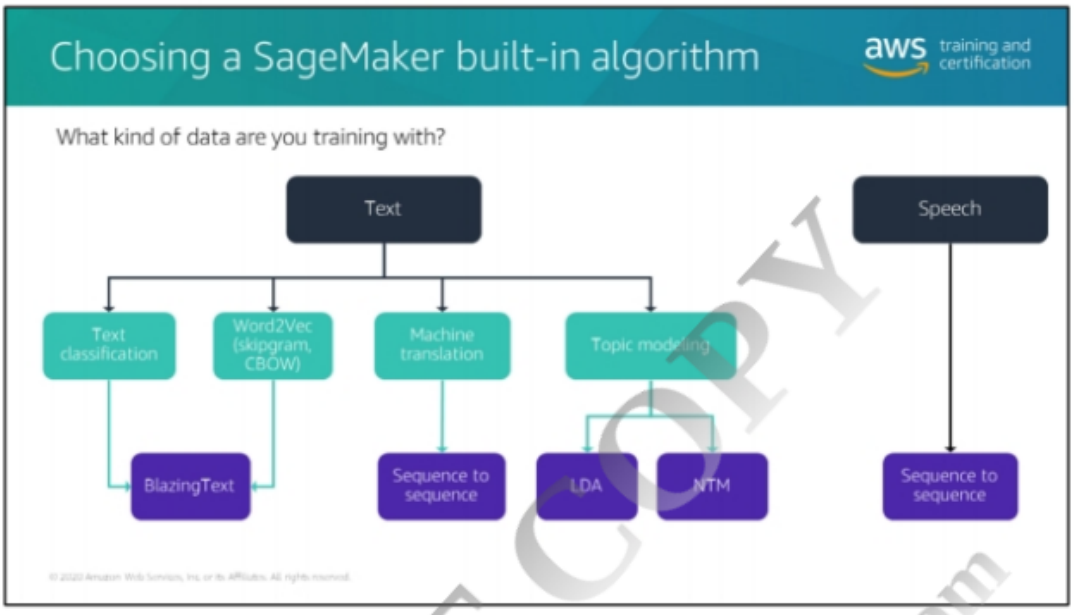
However, it's not always easy to figure out what algorithm you should use. There are different types of machine learning algorithms based on your use case and business requirements.

This is a general guide for choosing which algorithm to use depending on what business problem you have and what data you have. You should consider both when making this decision. Don't worry about memorizing these charts, but you might want to try printing them out or saving a copy of them, and keeping them somewhere easily accessible for use when you need it.

For more on these algorithms as they're used in Amazon SageMaker, refer to our documentation:
https://docs.aws.amazon.com/sagemaker/latest/dg/algos.html

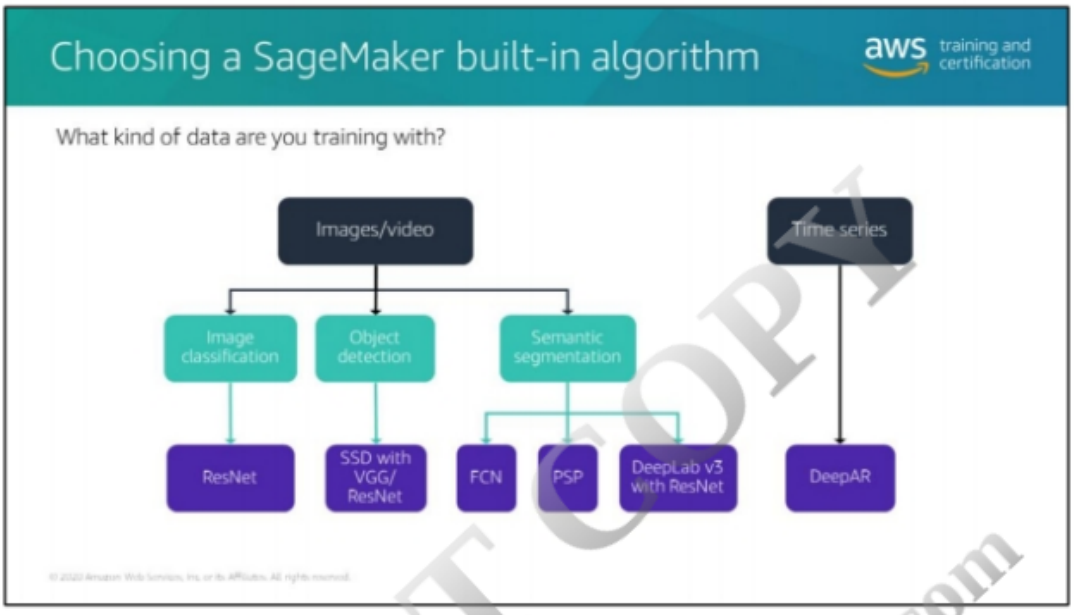


Here's another decision tree, this time for unsupervised learning problems.
Documentation: https://docs.aws.amazon.com/sagemaker/latest/dg/algos.html

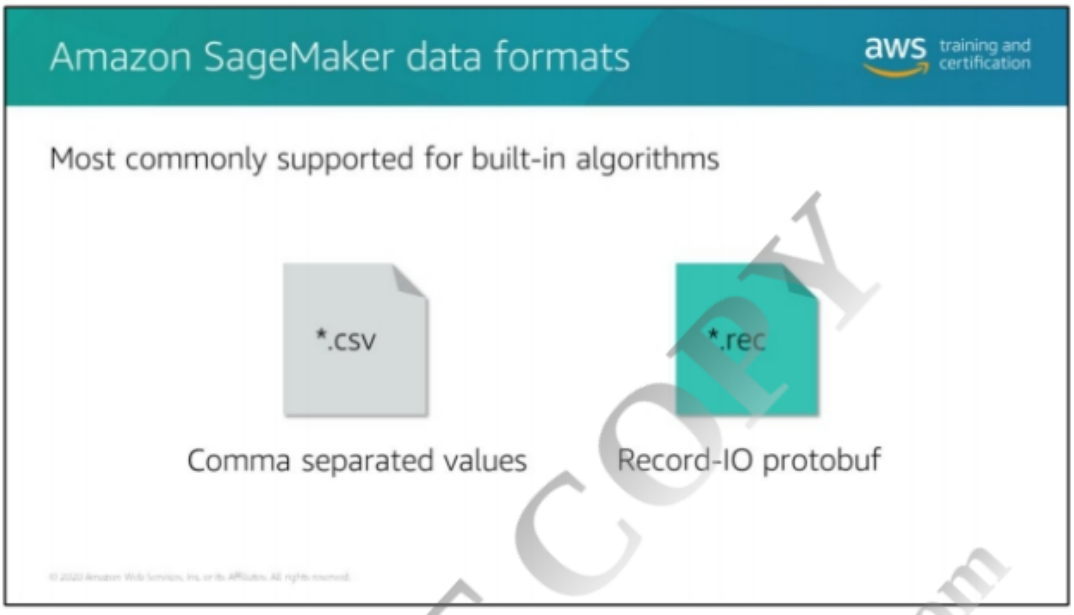


What kind of data you'll primarily be using can also influence which algorithm best fits your needs.

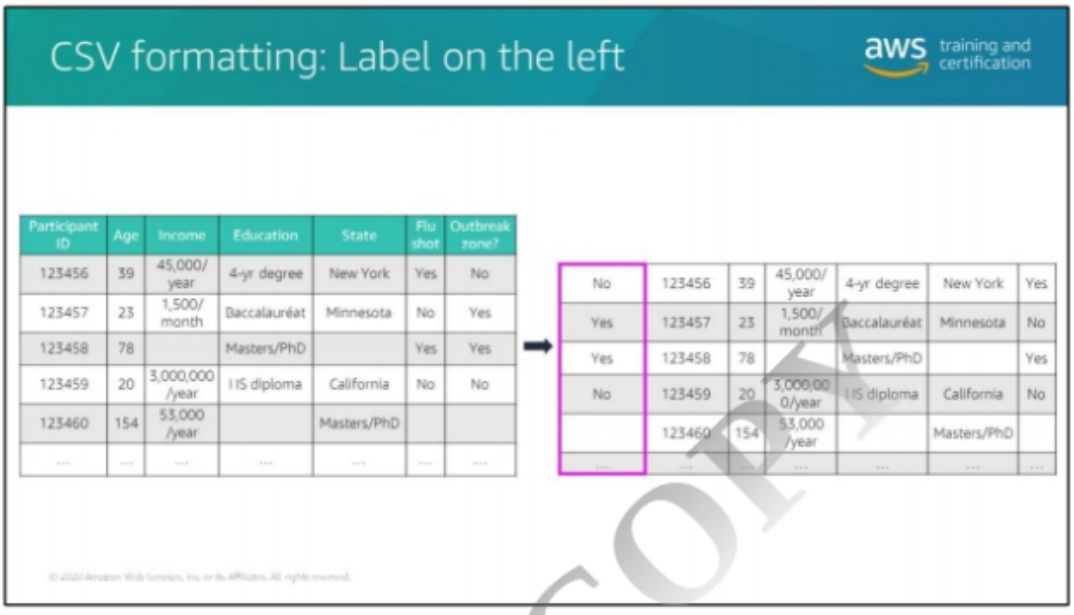
For example, if you're using raw text-based data, or recordings of speech.
Documentation: https://docs.aws.amazon.com/sagemaker/latest/dg/algos.html



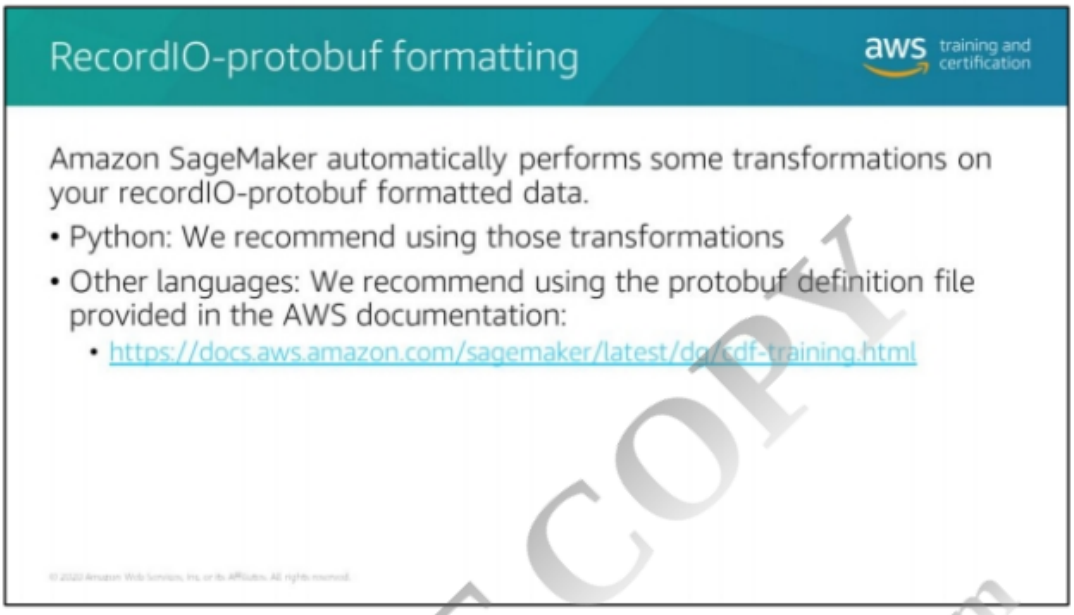
There are also algorithms available for using image, video, and time series data.
Documentation: https://docs.aws.amazon.com/sagemaker/latest/dg/algos.html



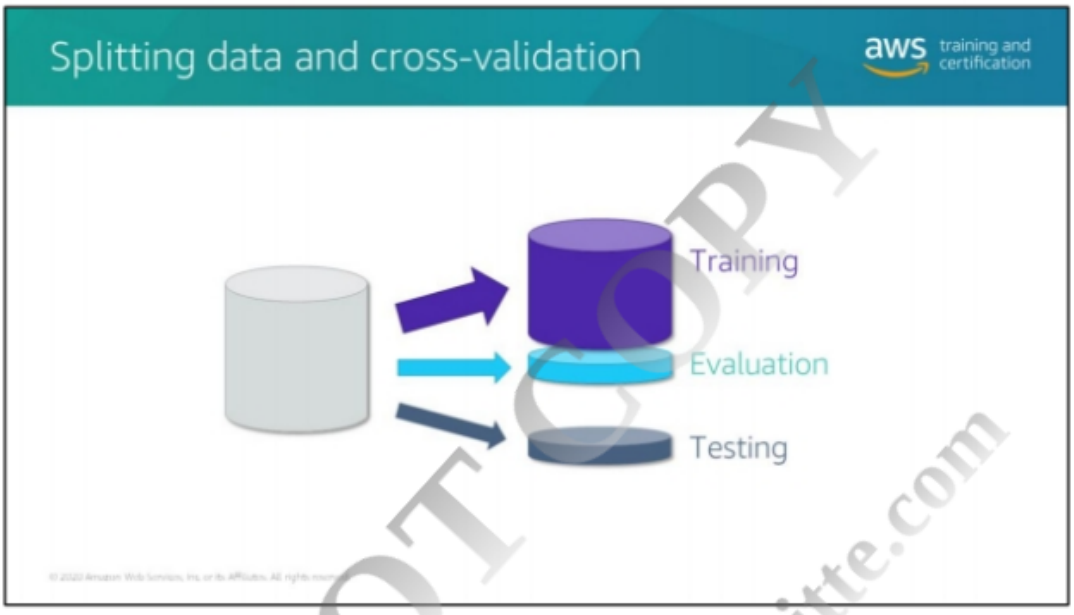
At this point, you've done a lot to clean and prepare your data, but that doesn't mean your data is completely ready to train the algorithm. Some algorithms require your data to be in a specific format. The two most common formats supported by the Amazon SageMaker built-in algorithms are CSV and recordIO-protobuf.



For Amazon SageMaker built-in algorithms, the label in your training dataset must be the first column on the left and your features should be to the right. Additionally, SageMaker requires that the CSV file have no header. However, some algorithms may not be able to work with training data in a dataframe format.

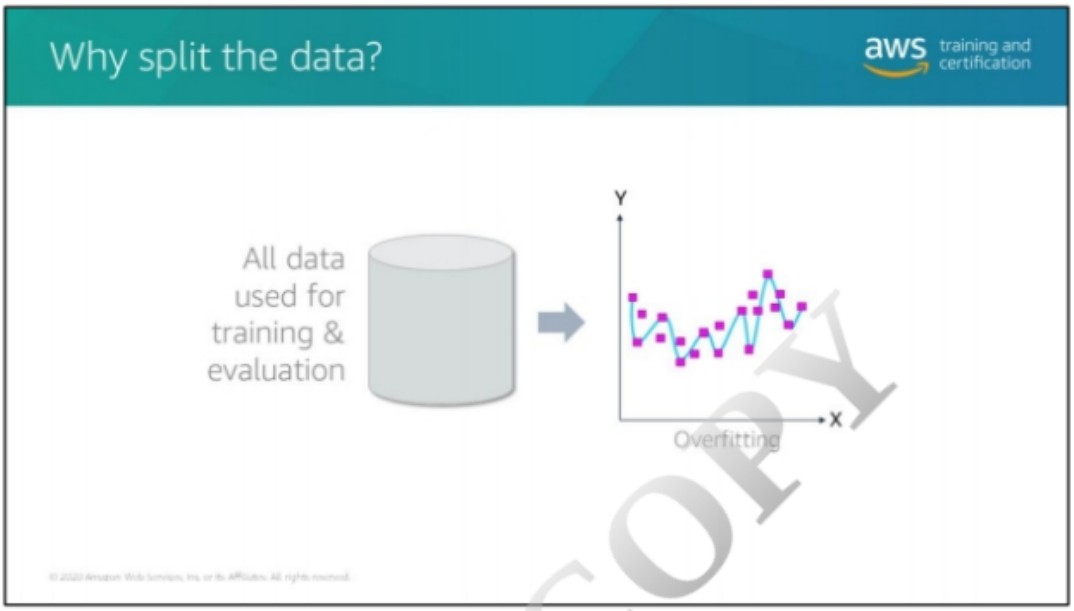


In the protobuf recordIO format, Amazon SageMaker automatically performs some transformations on your data. If you're using Python, we recommend you use those transformations; if you're using a different language, we recommend you use protobuf definition file that we provide in the AWS documentation here: https://docs.aws.amazon.com/sagemaker/latest/dg/cdf-training.html

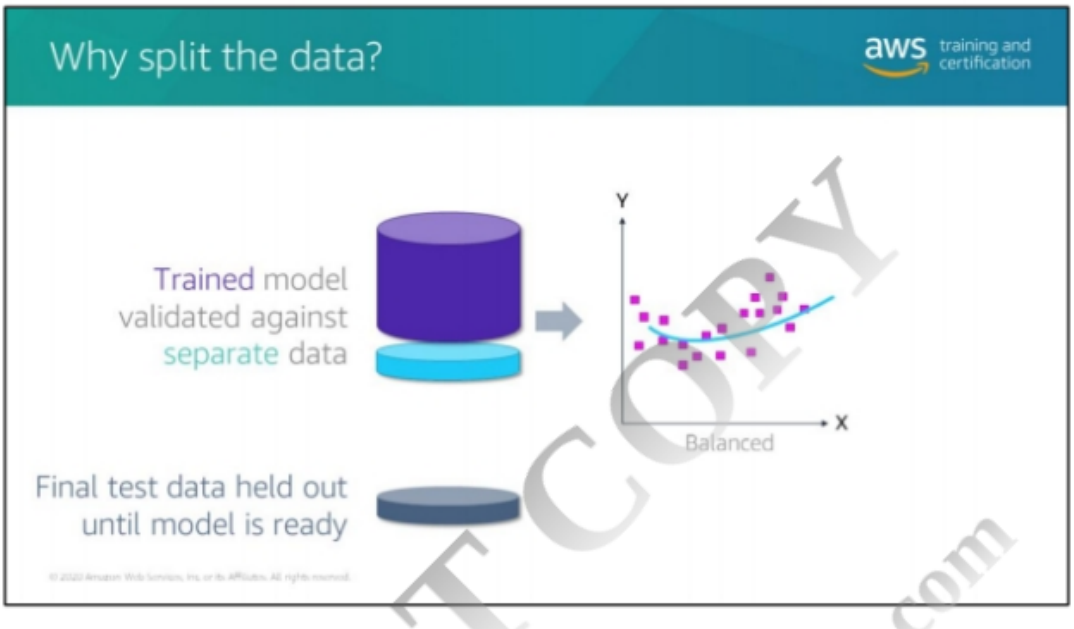


After preprocessing your data, you are almost ready to start training. But first, you need to decide how best to split up your data. Generally speaking, the goal of machine learning is to build a model that generalizes well. Put another way, you want your model to do well on both the data it already has and the data it does not have. This is why splitting your data is important.

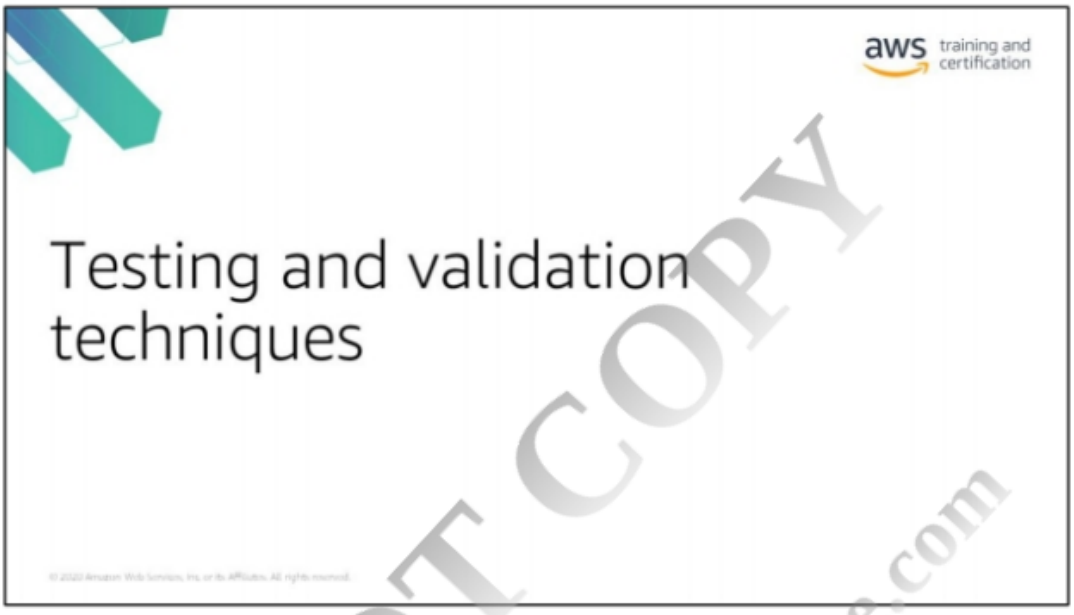
Splitting your data helps ensure that a chunk of your data qualifies as future production data that's similar to your training data, and that your model will predict with a similar accuracy as the data that your model has already seen. This will help the model be more generalizable.



Evaluating a model with the same data that it trained on will lead to *overfitting*, which is discussed in a later module. But for now, know that overfitting is where your model learns the particulars of a data set too well. It's essentially memorizing the training data, opposed to actually learning the relationships between features and labels so that the model can use what it learns from those relationships and patterns to apply to new data in the future.

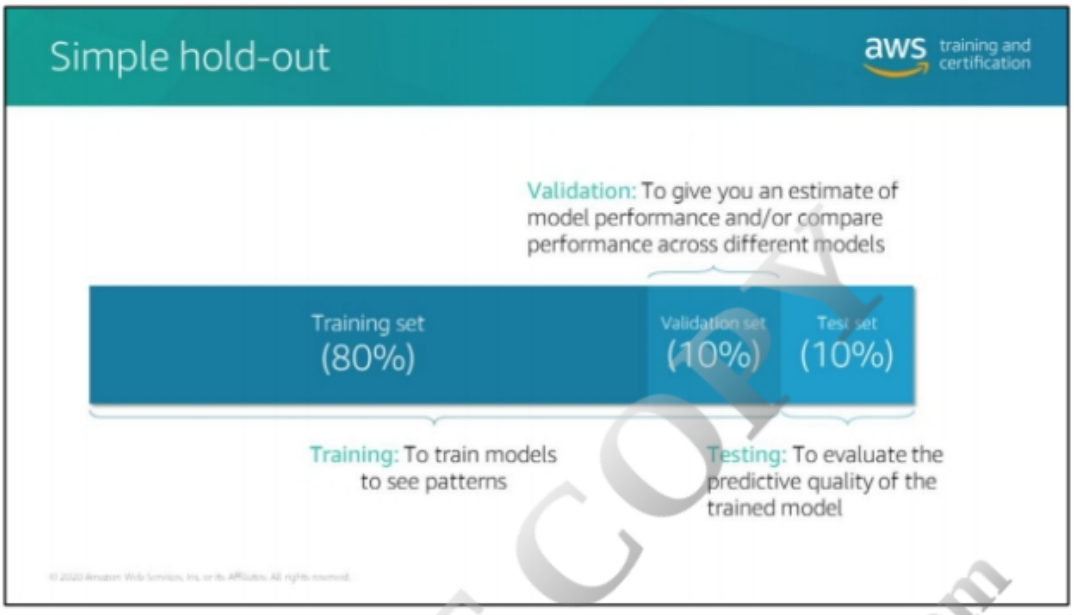


But generally, when you use some of the data to train your model and the rest of the data to evaluate, improve, and validate the model, you end up with a more balanced model that avoids overfitting to the training dataset.



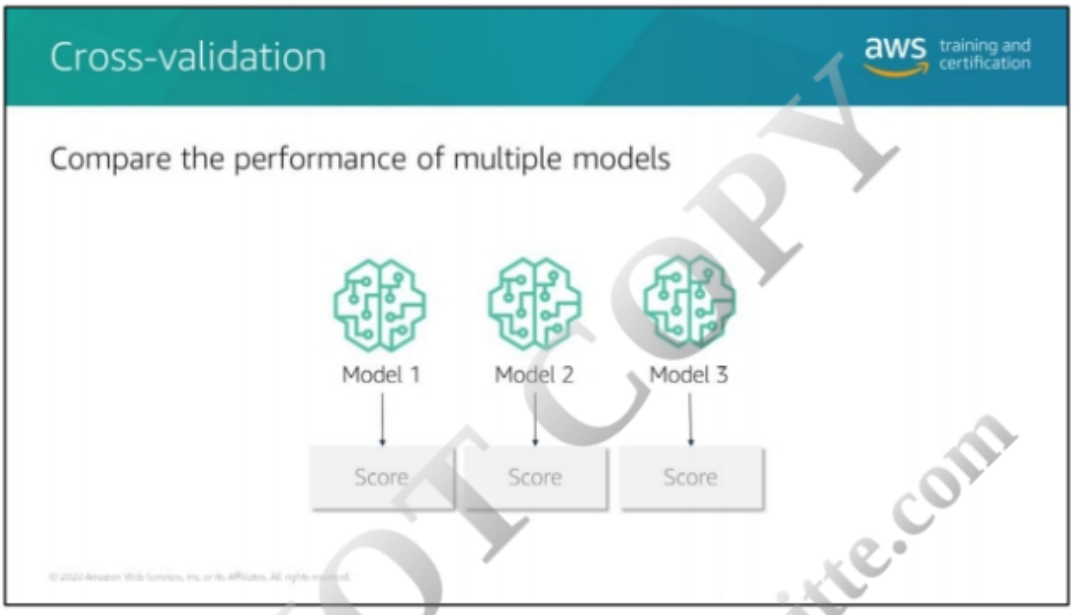
There are several techniques for mitigating overfitting and maximizing generalization. We'll talk about a few of them in this module.

- Simple Hold-Out Validation
- K-Fold Validation or cross-validation
- Leave-One-Out cross-validation
- Stratified K-Fold cross-validation
- Iterated K-Fold Validation with Shuffling

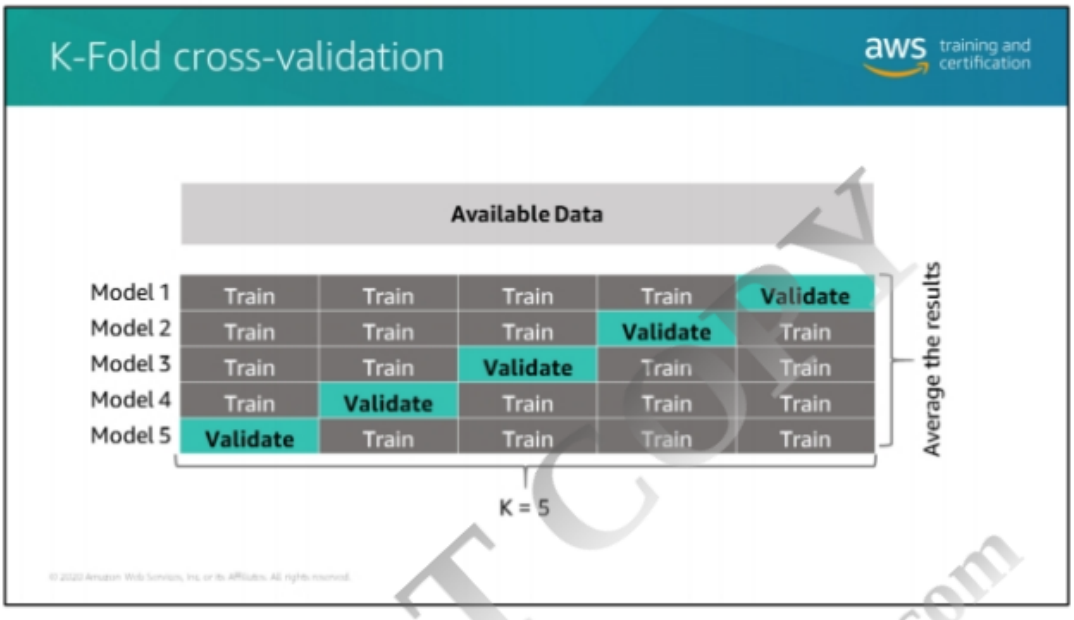


Simple hold-out is when you split your data into multiple sets, usually sets for training data, validation data, and testing data. *Training data*, which includes both features and labels, feeds into the algorithm you've selected to produce your model. The model is then used to make predictions over the *validation data set*, which is where you'll likely notice things you'll want to tweak and tune and change. Then, when you're ready, you run the *test data set*—which only includes features, since you want the labels to actually be predicted. The performance you get here with the test data set is what you can reasonably expect to see in production.

A common split when using the hold-out method is using 80% of the data for a training set, 10% for validation, and 10% for test. Or, if you have a lot of data, you can split it into 70% training, 15% validation, and 15% test.



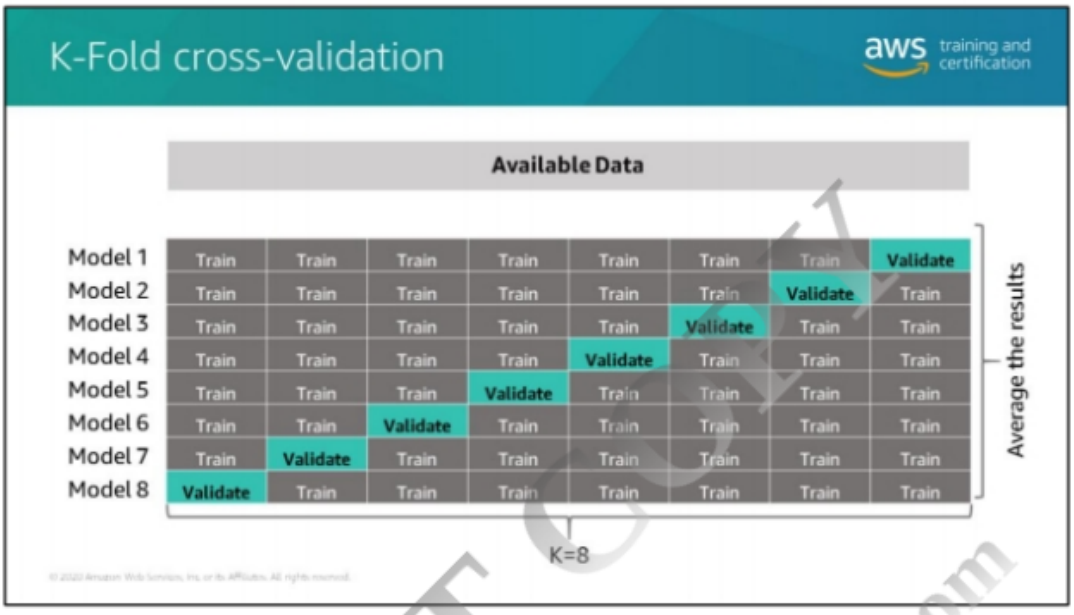
You can also use cross-validation methods to compare the performance of multiple models. The goal behind cross-validation is to help you choose the model that will eventually perform the best in production.



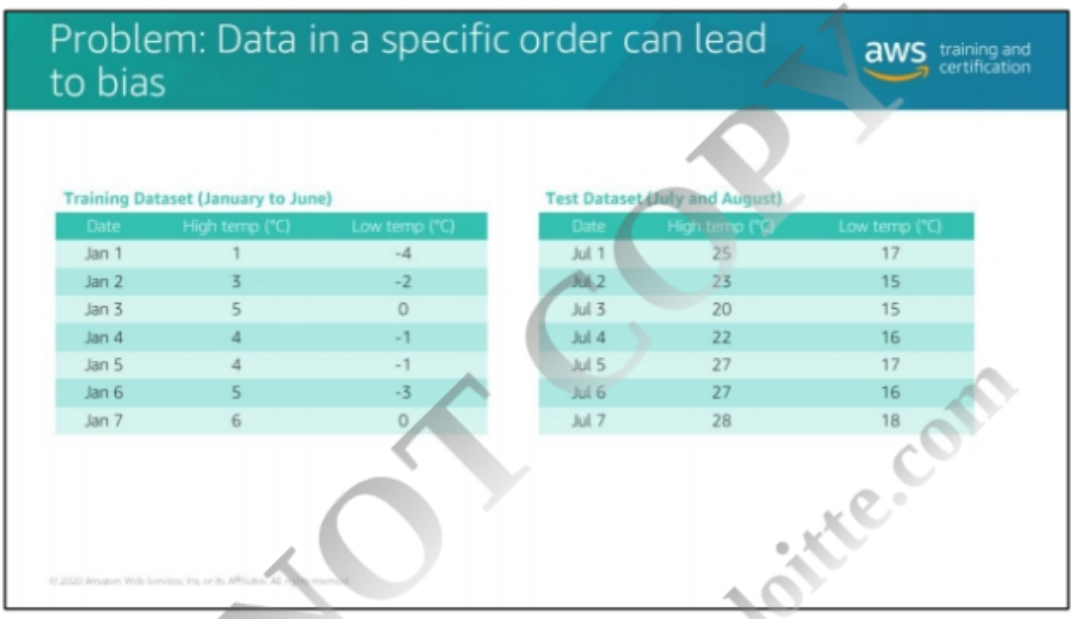
So for a small dataset, we can use K-Fold cross-validation to use as much of the data as possible, while still having relatively good metrics, in order to choose which model is better. K-Fold cross-validation randomly partitions the data into K different segments. For each segment, we'll use the rest of the data outside of it for training in order to do a validation on that particular segment.

We do this for each of those K segments. Essentially, we are applying different models to different pieces of the validation dataset in order to have some idea about how good the models perform. At the end of the day, we are using all of the data for training and the cross-validation results are averaged to give us an idea about the performance of the model. Different machine learning models can be applied to the training data using K-Fold cross-validation enabling us to compare the performance of the models based on the averaged result.

Let's look at an example. Here we have a 5-Fold cross-validation; the available training data is separated into five different chunks. For the training of the first model, we are using all those chunks as the training data and then we're going to calculate the metrics on this test piece. For the second model we are going to use these pieces as training. After the model is trained, you apply it to this validation piece. We do the same thing five times. We use all the training data and we test it on five different models on different chunks of the validation data - eventually testing it on all data points.

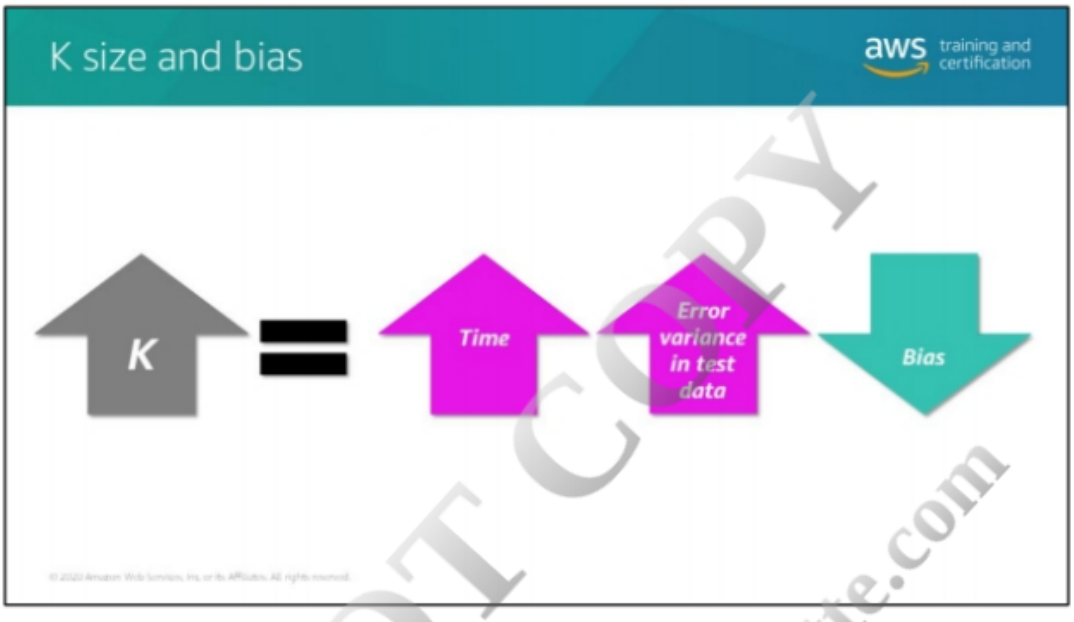


The prior example was a 5-Fold cross-validation, and this is what a 8-Fold cross-validation would look like. The data set you're training on each time is smaller, but you're performing more tests.



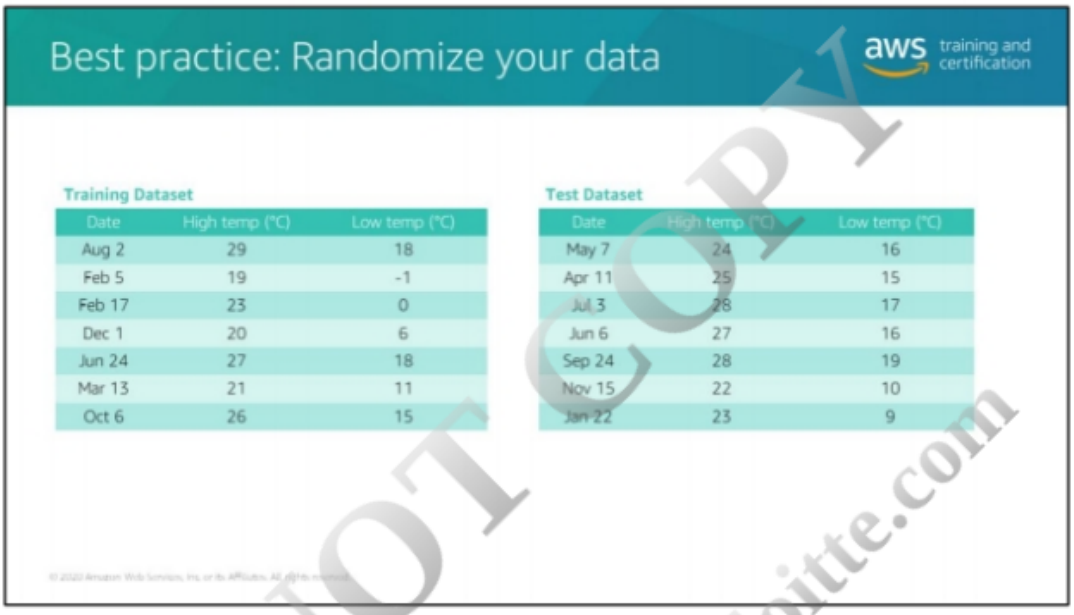
One other thing to note about splitting your data: Data in a specific order can lead to biases in your models. This is especially true if you're working with structured data. If your data is in a specific order—for example, if there are dates listed sequentially—your model gets used to that structure and will adapt this pattern as it learns. Eventually, when you run your model against your test data, this patterning of sequential dates will be applied, biasing the model.

In this example, the data is in order of date, and since it's measuring temperatures, that leads to very different sets of data depending on which part of the calendar year is being sampled. So a model that's trained on January's data will run into problems when validating it against July's data, since it's so heavily biased towards January.

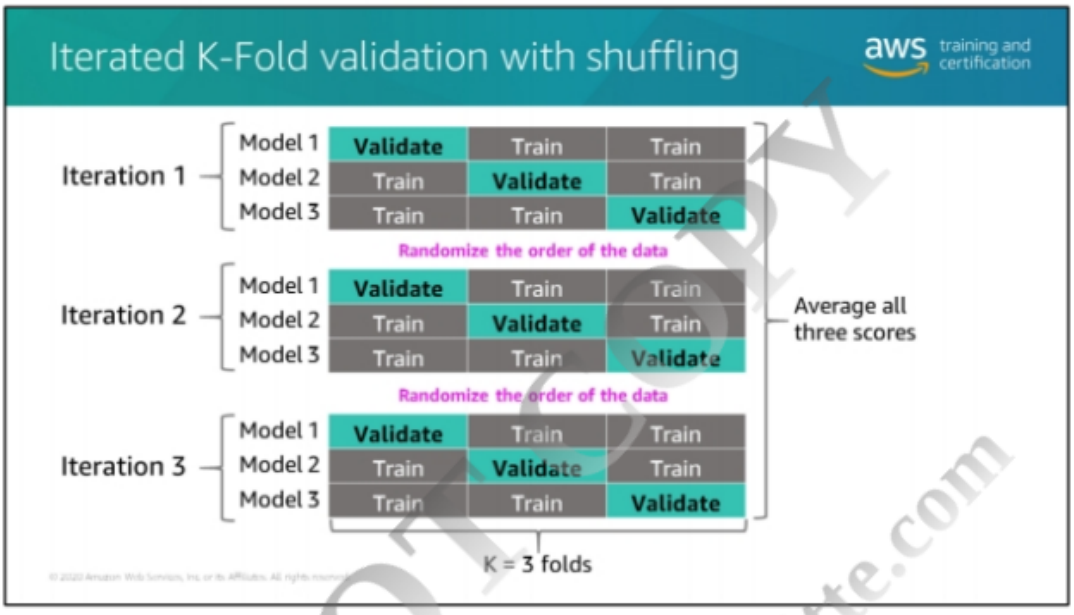


If we decide to go with a larger K, we're going to train the model more times, effectively using all the training data every time. So, a larger K definitely means more time and more variation in the test error that you get for every subset of the dataset because of the size of the test dataset will keep getting smaller and smaller. On the other hand the data that you will use for training will be larger for a large K so the bias will reduce. For a smaller K we are using smaller chunks of the data and we're training the model fewer times. So smaller Ks are going to be more biased because we're using smaller chunks of the original training dataset. Biased in this sense means there's a systematic difference between the true model and the estimated model. Like variance, we'll talk more about bias in a following module.

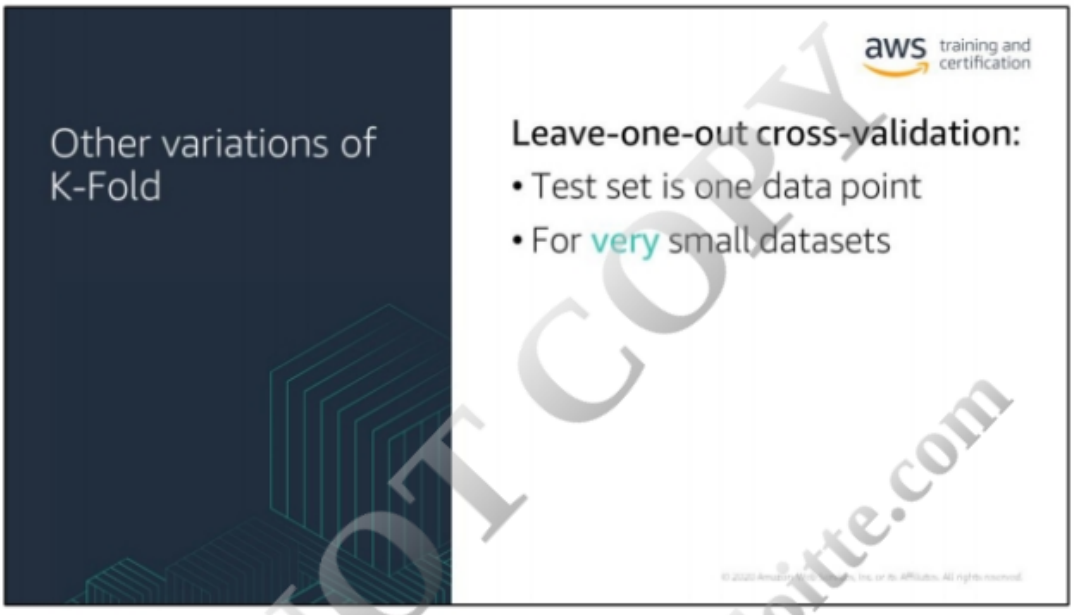
Typically, when training machine learning algorithms, we start with a number between five and ten folds to see the performance. Those numbers can then be changed based on your specific business problem and needs.



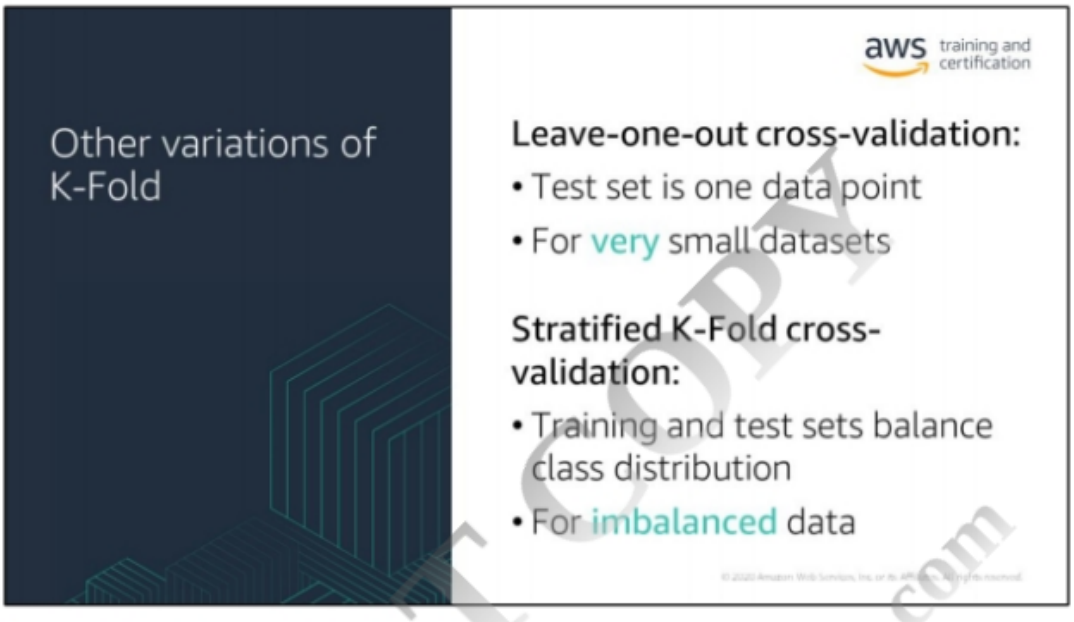
That's why we recommend randomizing your data during your split to help your model avoid bias.



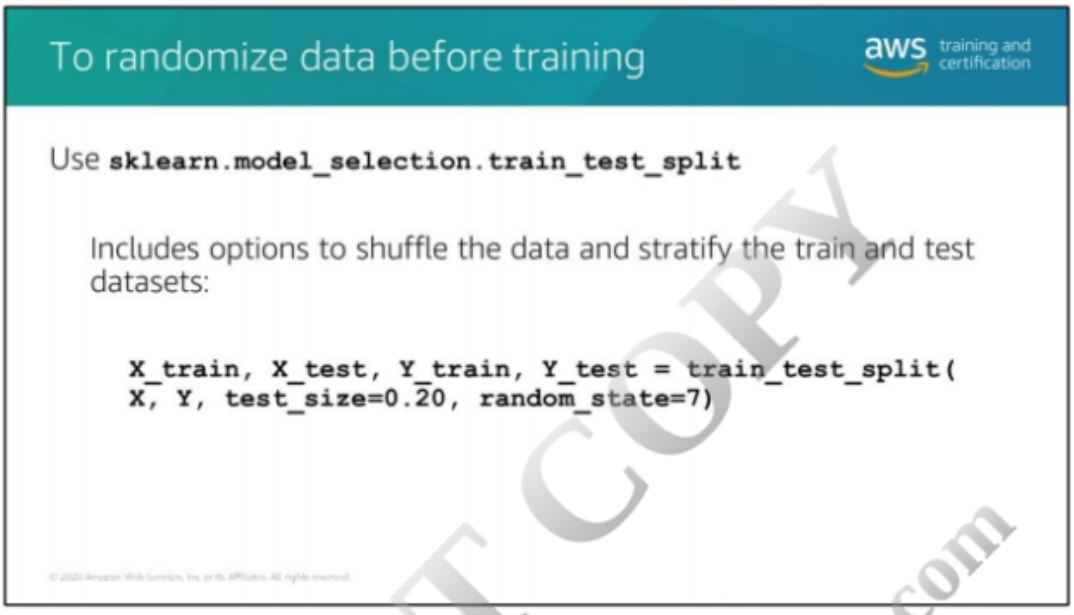
Iterated K-Fold Validation with shuffling is another variation of K-Fold Validation. It consists of applying K-fold validation multiple times, shuffling the data every time before splitting it K ways. The final score is the average of the scores obtained at each run of K-fold validation.



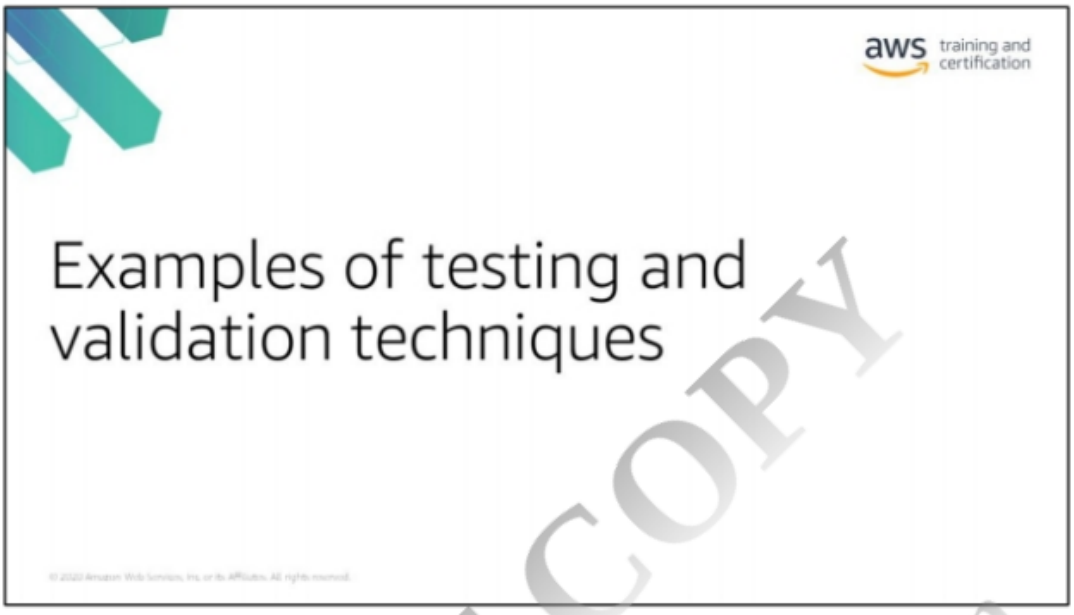
There may be some variations of K-Fold cross-validation, for example, the Leave-One-Out cross-validation. In the Leave-One-Out cross-validation, the K is equal to N. Every time we leave one data point out for testing, we are using the rest in the training data. This is usually used for very small datasets where every data point is very valuable.



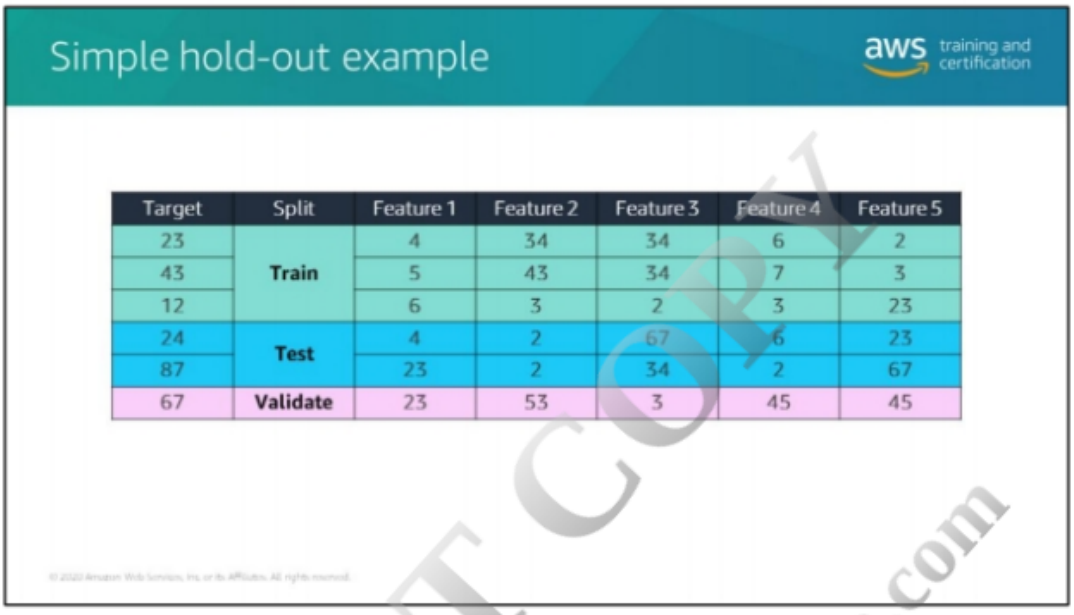
There's also stratified K-Fold cross-validation, which is often used when there are seasonalities or subgroups in small proportion in the data set. Stratified K-Fold cross-validation is going to ensure that for each fold, there are some equal weight proportions of the data for every different fold. For instance, while splitting the data you might want to ensure that there is an equal representation of a certain target variable among the different folds.



Forgetting to randomize your data is a costly and unfortunately common mistake. This may cause the training and test datasets to have a different distribution due to features that might be time dependent or season dependant. Thankfully, you can use Sklearn to automatically split and shuffle the data at the same time.

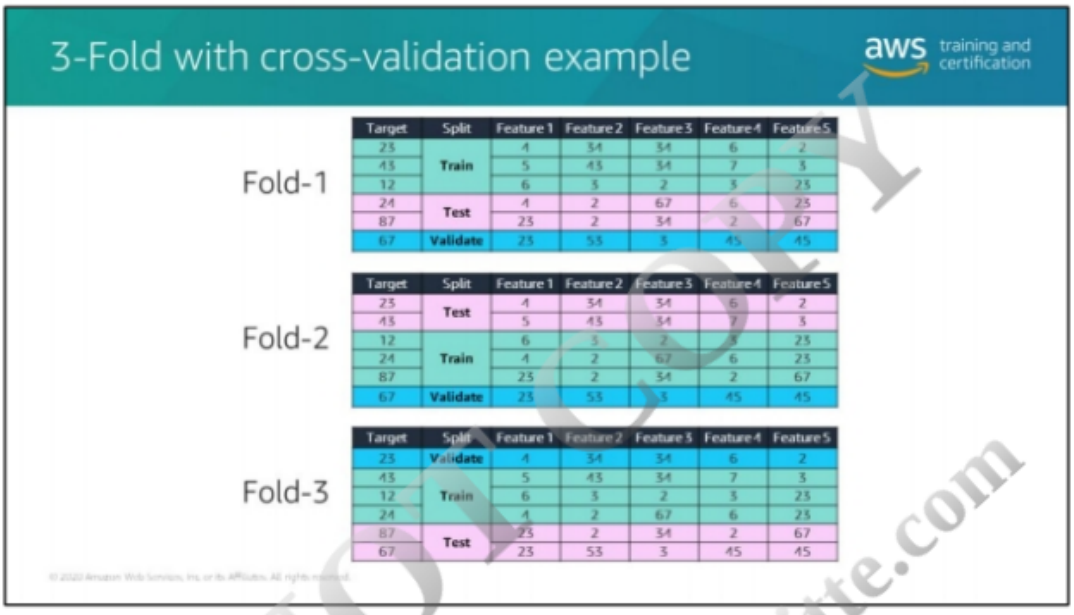


Let's wrap this section up by looking at an example of dataset and three different ways you can split it up using some of the methods we just discussed.



Target	Split	Feature 1	Feature 2	Feature 3	Feature 4	Feature 5
23	Train	4	34	34	6	2
43		5	43	34	7	3
12		6	3	2	3	23
24	Test	4	2	67	6	23
87		23	2	34	2	67
67	Validate	23	53	3	45	45

Here's the data split using the simple hold-out method.

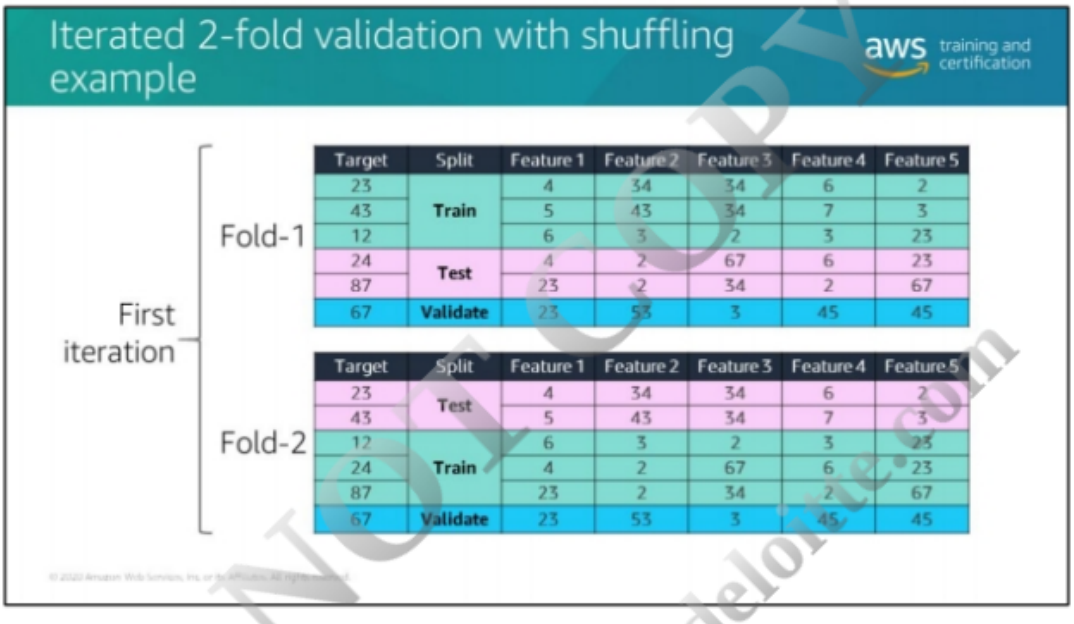


Target	Split	Feature 1	Feature 2	Feature 3	Feature 4	Feature 5
23	Train	4	34	34	6	2
43		5	43	34	7	3
12		6	3	2	3	23
24	Test	4	2	67	6	23
87		23	2	34	2	67
67	Validate	23	53	3	45	45

Target	Split	Feature 1	Feature 2	Feature 3	Feature 4	Feature 5
23	Test	4	34	34	6	2
43		5	43	34	7	3
12	Train	6	3	2	3	23
24		4	2	67	6	23
87		23	2	34	2	67
67	Validate	23	53	3	45	45

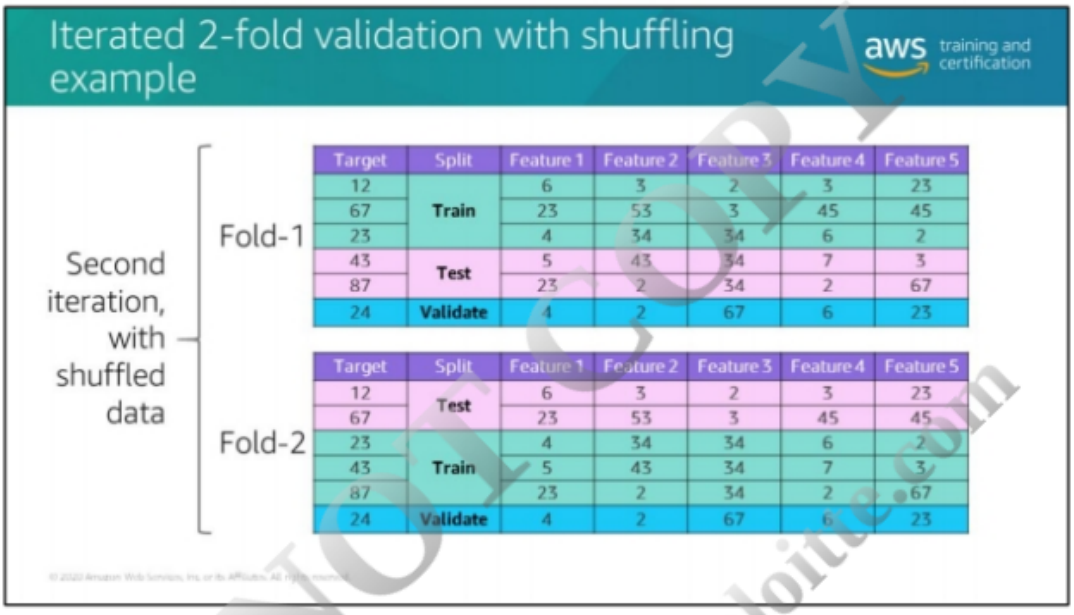
Target	Split	Feature 1	Feature 2	Feature 3	Feature 4	Feature 5
23	Validate	4	34	34	6	2
43	Train	5	43	34	7	3
12		6	3	2	3	23
24		4	2	67	6	23
87	Test	23	2	34	2	67
67		23	53	3	45	45

And here it is using K-Fold cross-validation with three folds.

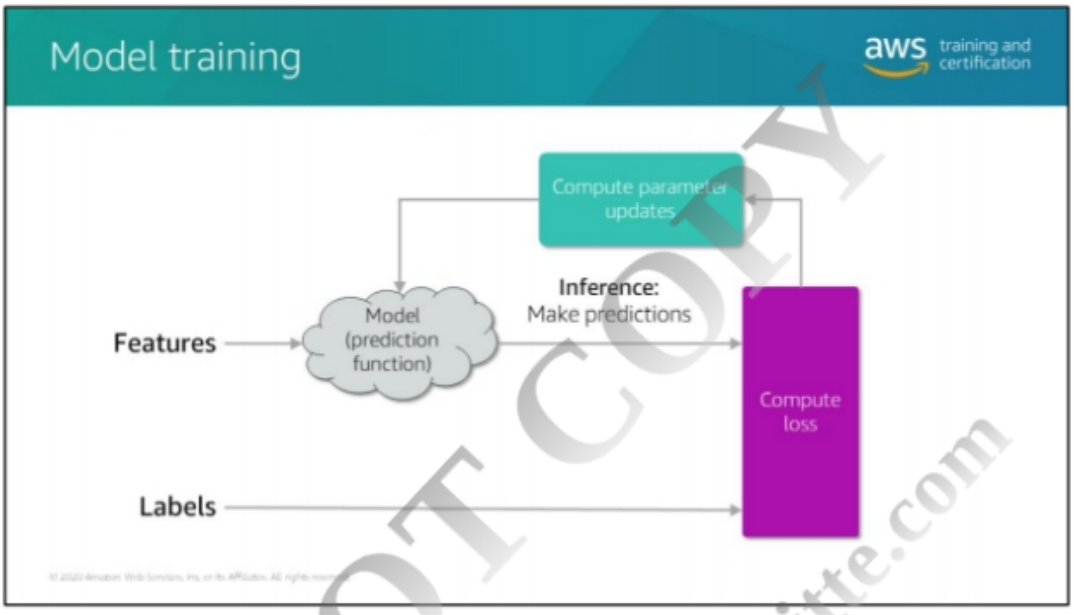


And finally, here it is using Iterated K-Fold Validation with Shuffling. In this case, we're using two folds and two iterations.

Here's the first iteration.



And now the second iteration. Remember to shuffle the data before folding it again.

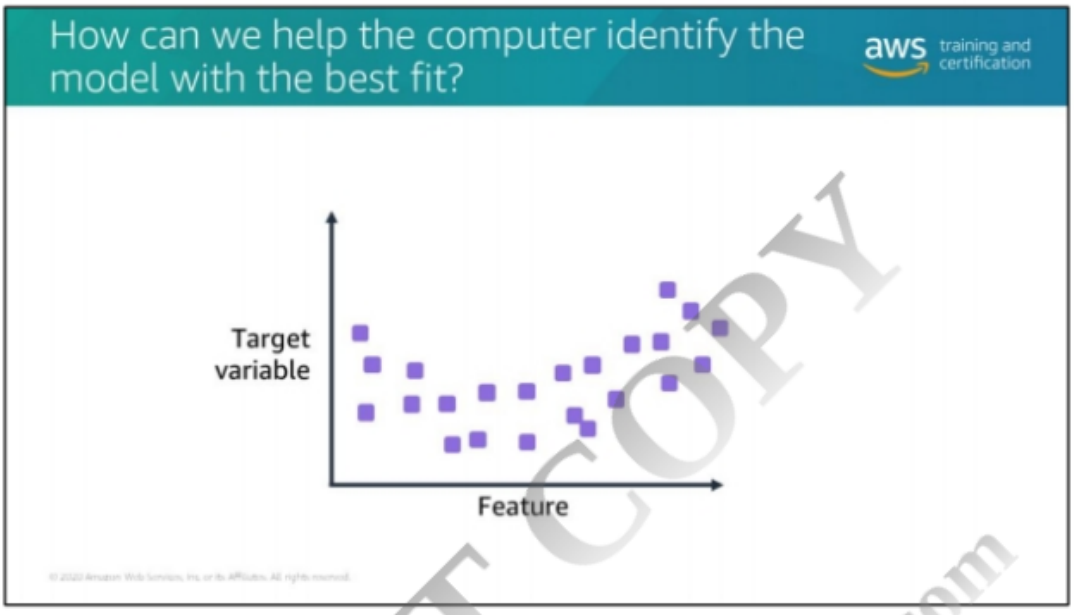


So we're officially ready for model training. But let's look deeper into what actually happens during that process.

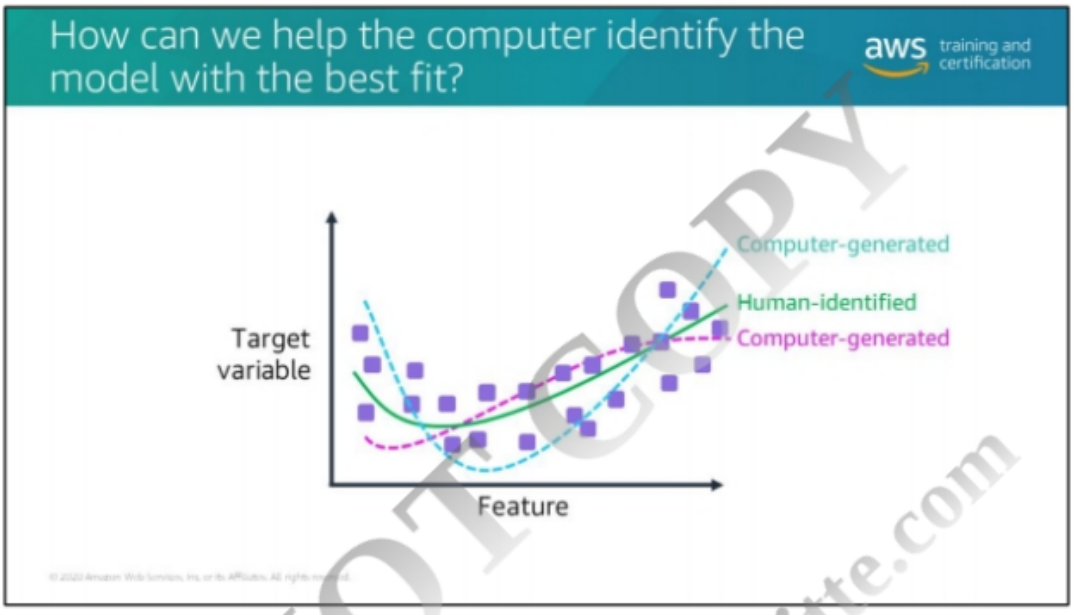
During training, the machine learning algorithm updates a set of numbers known as *parameters* or *weights*. The goal is to update the parameters in the model in such a way that the computed or predicted output becomes as close as possible to the true output (as seen in the data).

This can't be done in one iteration, because the algorithm has not learned yet; it has no knowledge of how changing weights will shift the output closer towards the expected value. Therefore, it watches the weights and outputs from previous iterations and shifts the weights to a direction that lowers the error in generated output. This iterative process stops either when a defined number of iterations have been run or when the change in error is below a target value.

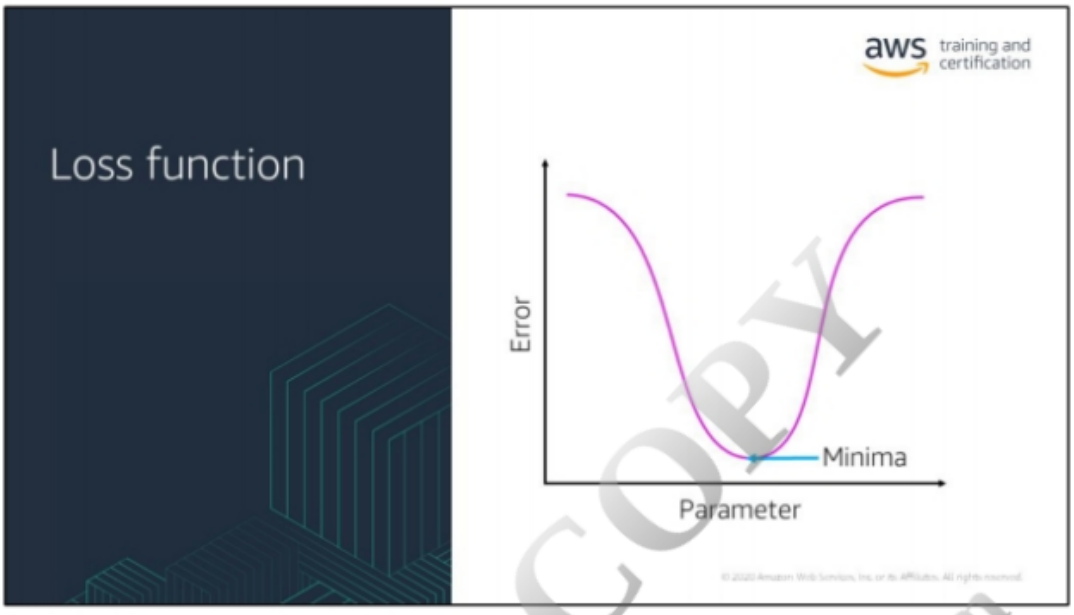
If the error in the output gradually decreases with each successive iteration, the model is said to converge and the training is considered successful. If on the other hand, the errors either increase or change randomly between iterations, the assumption in building the model needs to be re-evaluated.



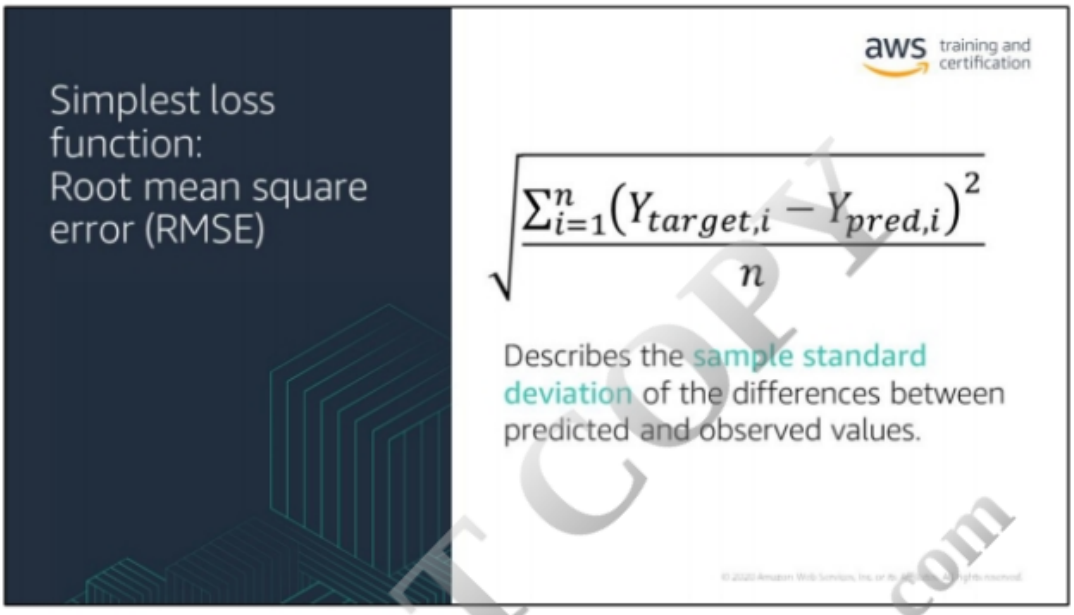
Let's take a simple example of a linear regression model. Assume that you have about 1,000 data points with one feature column and one prediction column. They are distributed as seen in the figure on the slide. Let's call the feature column X and the prediction or label column Y. During training, remember, the model is trying to find a way to map X to Y, the simplest way possible.



For a human, this figure is easy to understand. Just by looking at it, you can draw a line through it representing the best possible fit for this data. But for an ML model, this process needs a couple of components to achieve the same results: a loss function and an optimization technique.

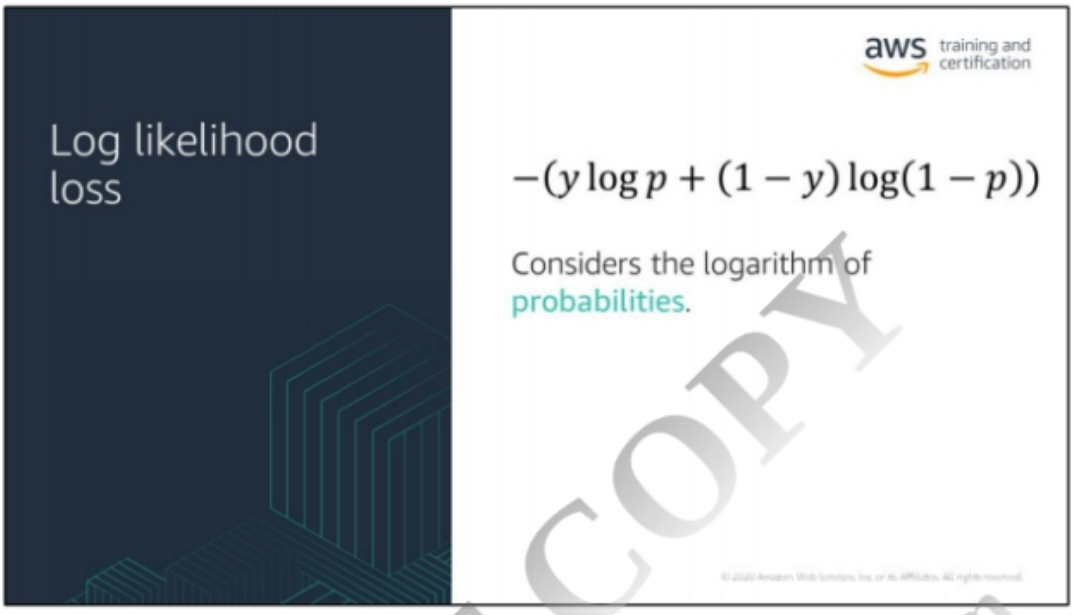


The *loss function*, which is sometimes called the *objective function,* is the measure of error in your model's predictions given a set of weights. At any given iteration of training, the model produces an output prediction value, using the weights as calculated up to that point. This output would almost always deviate from the true output to a certain extent. This difference is the loss function. The loss function is the metric used to update the weights after each iteration.



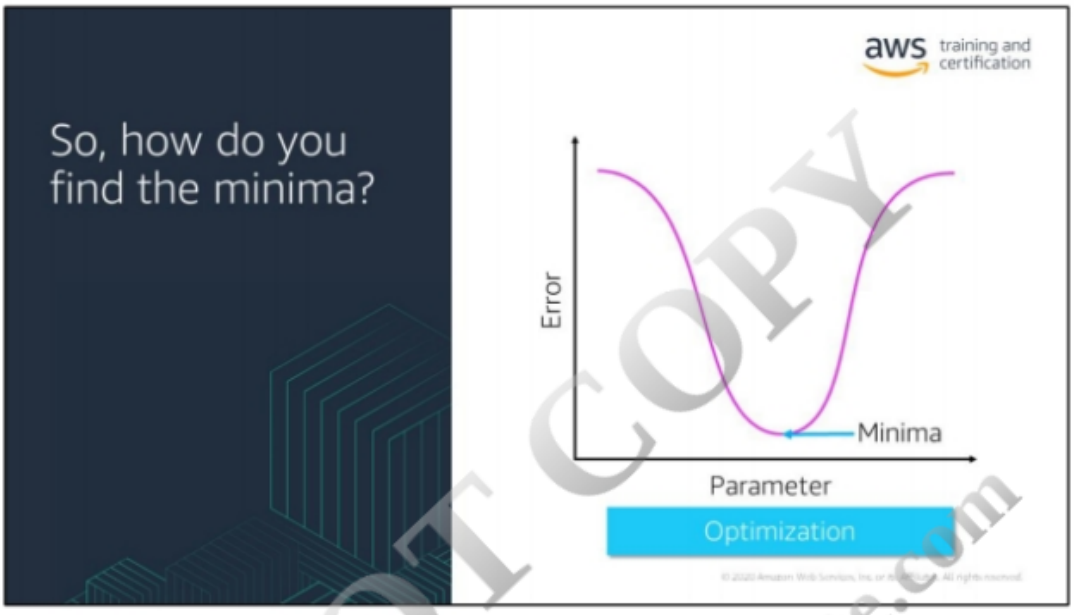
There are various ways this loss (or error) can be calculated. The simplest form of loss function is known as *Root Mean Square Error* (RMSE). The RMSE describes the sample standard deviation of the differences between the predicted and observed values. This is calculated by taking the difference between true output and calculated output, squaring it, and averaging it out across the whole dataset.

In the equation, Y_{target} is the target output and Y_{pred} is the predicted output for i-th observation.

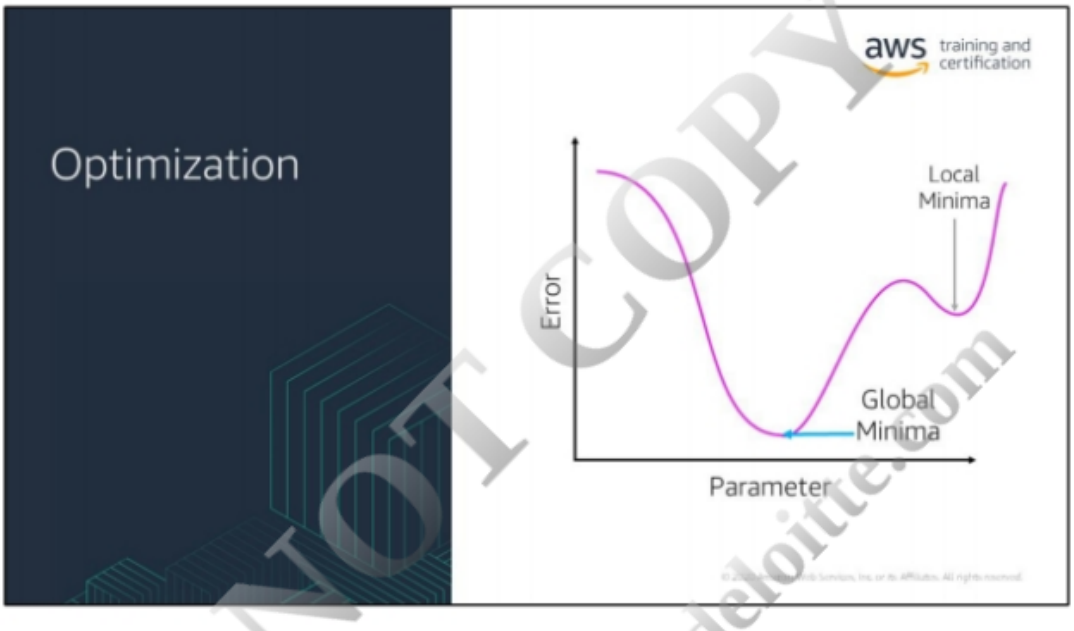


A common variation of loss function is *log likelihood loss*, also known as *cross-entropy loss*. With log likelihood loss, instead of the raw probabilities of predictions of each class, the logarithm of probabilities is considered. For binary classification, the formula for log likelihood loss becomes this equation, where y is a binary indicator (0 or 1) of whether the class label is the correct classification for the observation, and p is the model's predicted probability that the observation belongs to that class.

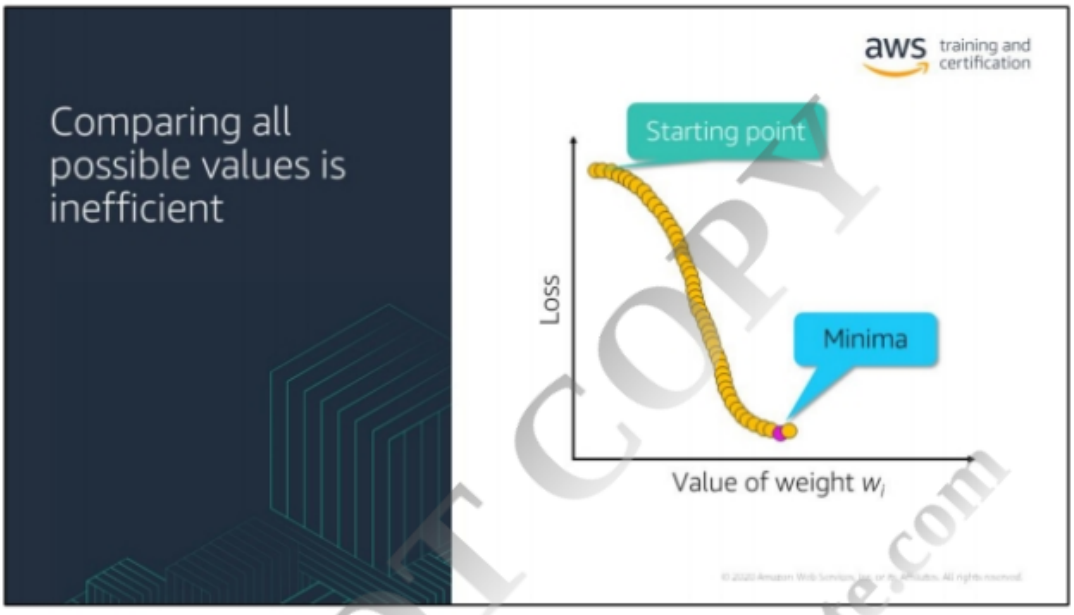
But how exactly does the error get minimized during model training? How are the weights updated to achieve the minimum amount of loss in the model?



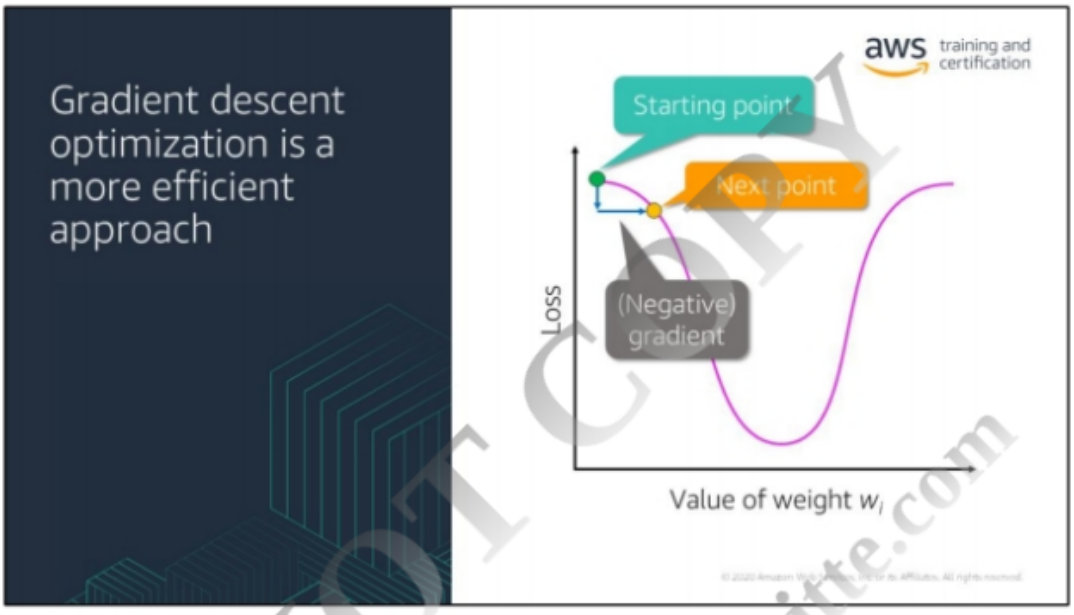
This is where optimization techniques come in. An *optimization technique* seeks to minimize the loss function that quantifies this penalty or error as a single value. Take for example this 1 dimensional graph (1D). The error is given on the Y axis and the single feature is given on the X axis. Just by looking at the graph you know that you should always have the lowest loss possible at the bottom of that curve. But for a machine learning algorithm, it doesn't know this graph for that particular feature so it needs a way to learn how to traverse to the bottom of this well, also known as trough.



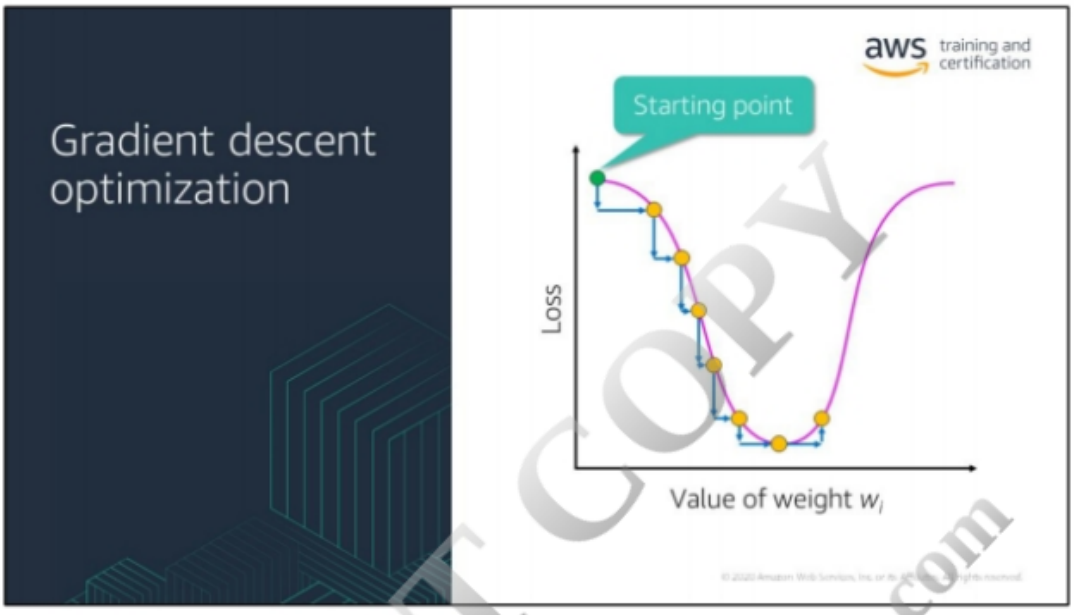
The loss function when plotted for a single feature is a curve (or plane in 2Dimension for 2 features, or hyper-plane in general), and has its own hills and troughs. The goal is to move to the lowest point (trough) within this landscape, representing a particular value of weight for which the loss is minimum. Any loss surface used in training a machine learning model will include a lowest trough called a *global minima*. Often, machine learning models are known to fall into what are called *local minima,* which prevents the model from improving more and thus preventing them from reaching the global minima.



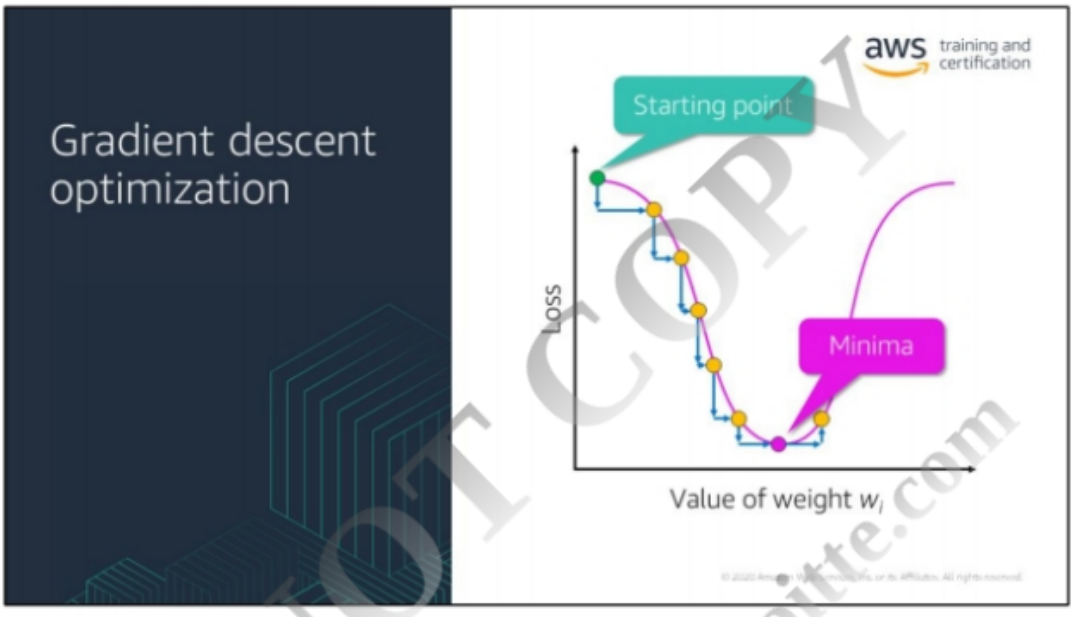
It is very inefficient to attempt to find the minimum by taking all possible value combinations of the weights and comparing the values. Instead optimization is used, which is at the heart of a lot of ML algorithms.



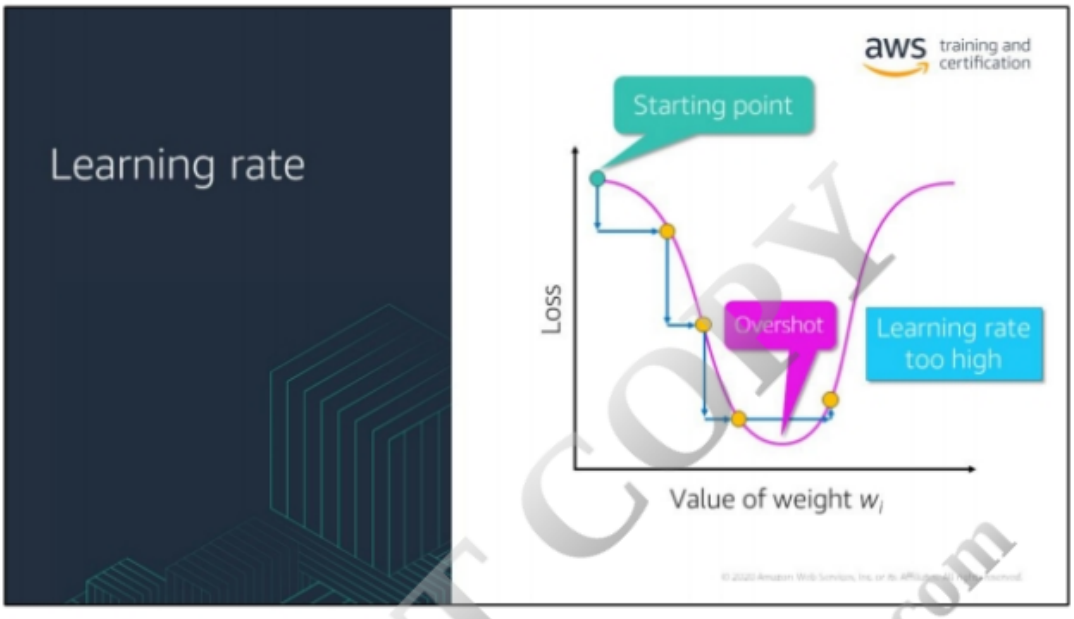
Let's look at one of the simplest optimization techniques known as *gradient descent*. In this approach, the loss function is calculated at that step using the complete dataset, and the *slope* (also known as *gradient*) of the error curve is calculated using the loss function value at the current point. Conceivable, for gradient descent, the slope at any point may point us towards the bottom most point of the surface. Therefore, the weights are updated such that the calculated error moves by a small step towards the direction pointed by the slope (gradient).



As the loss decreases, you get closer to the optimal value of the weight.



Eventually, the minima is approximated when enough of these operations have been performed to determine the weights that result in the least amount of loss.



The size of the step is crucial, because taking a large step might cause the model to overshoot the bottom most point and cause the model to swing back and forth and never reach the minimum.

